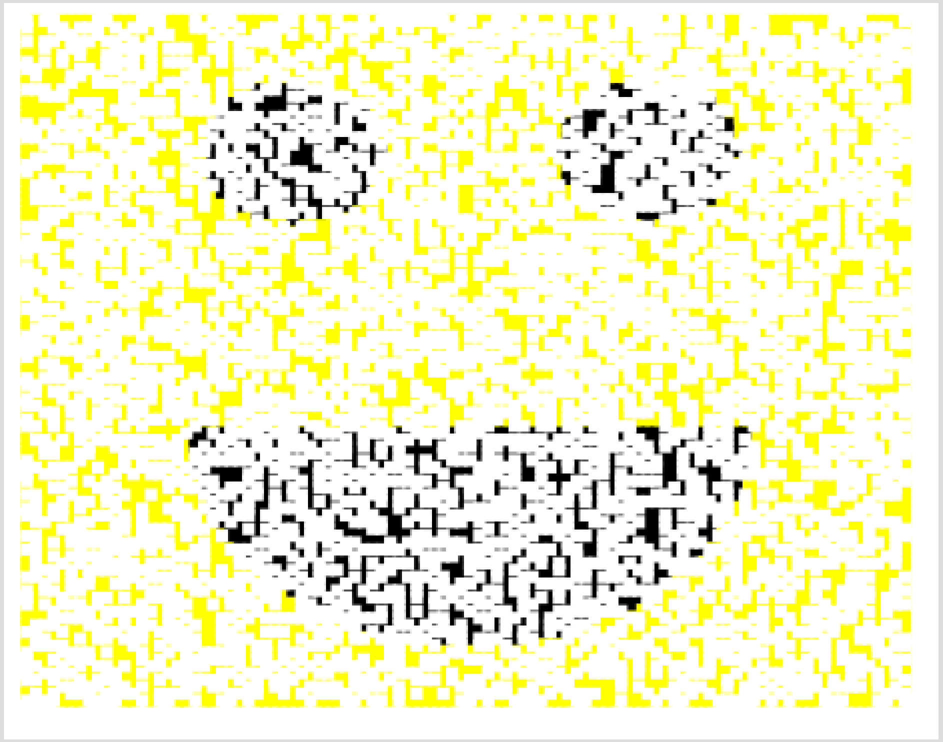
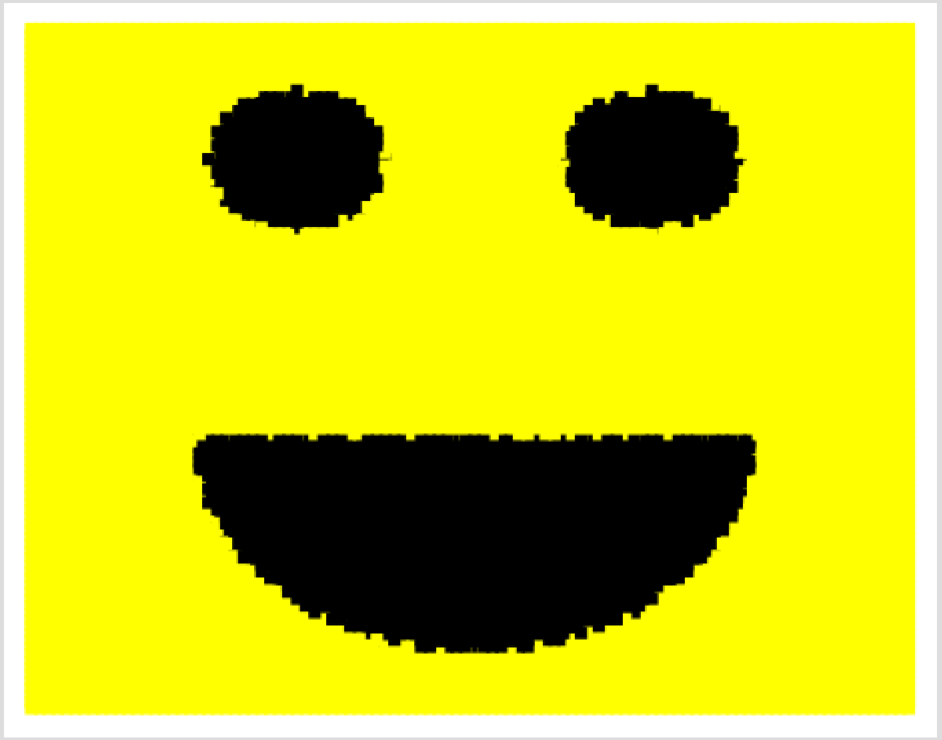
K-nearest neighbors (Ch. 18.8)





Announcements

Midterm2 grades up on gradescope

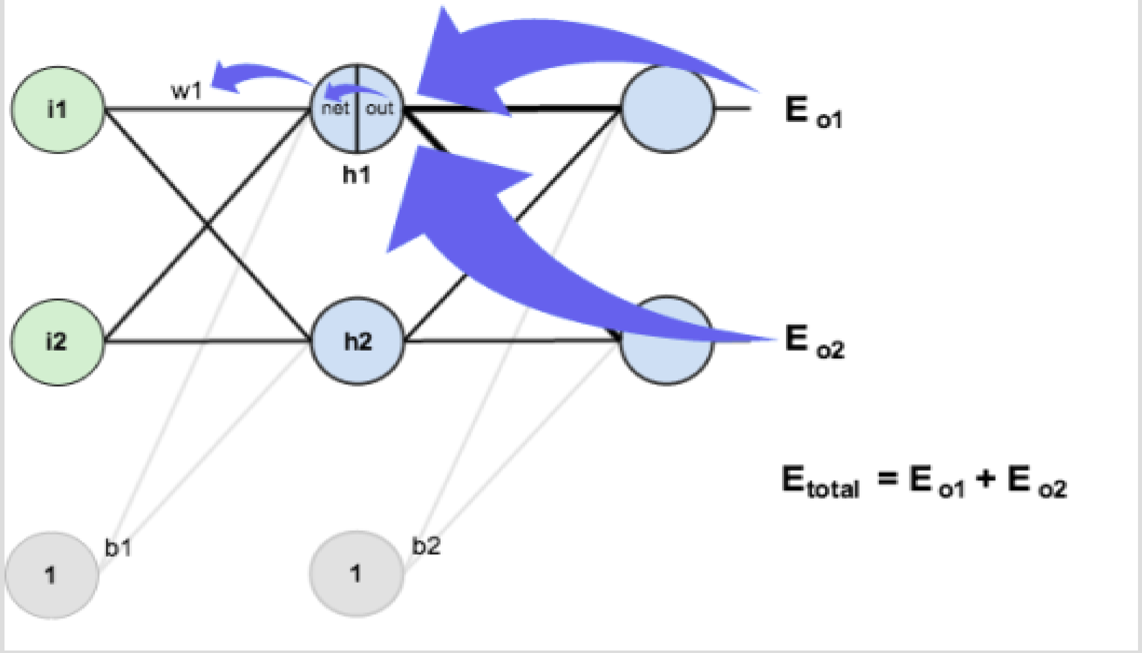
Review: Back-propagation

For w_1 it would look like:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



(book describes how to dynamic program this)

Learning types

Neural networks are what we call a <u>parametric model</u>, (not to be confused with the statistics definition) as the inputs are fixed

What we will talk about today (k-nearest neighbor) is a <u>non-parametric model</u> as it will not fix the input space (no assumptions about parameters from examples in model)

What type were our decision trees?

Non-parametric model

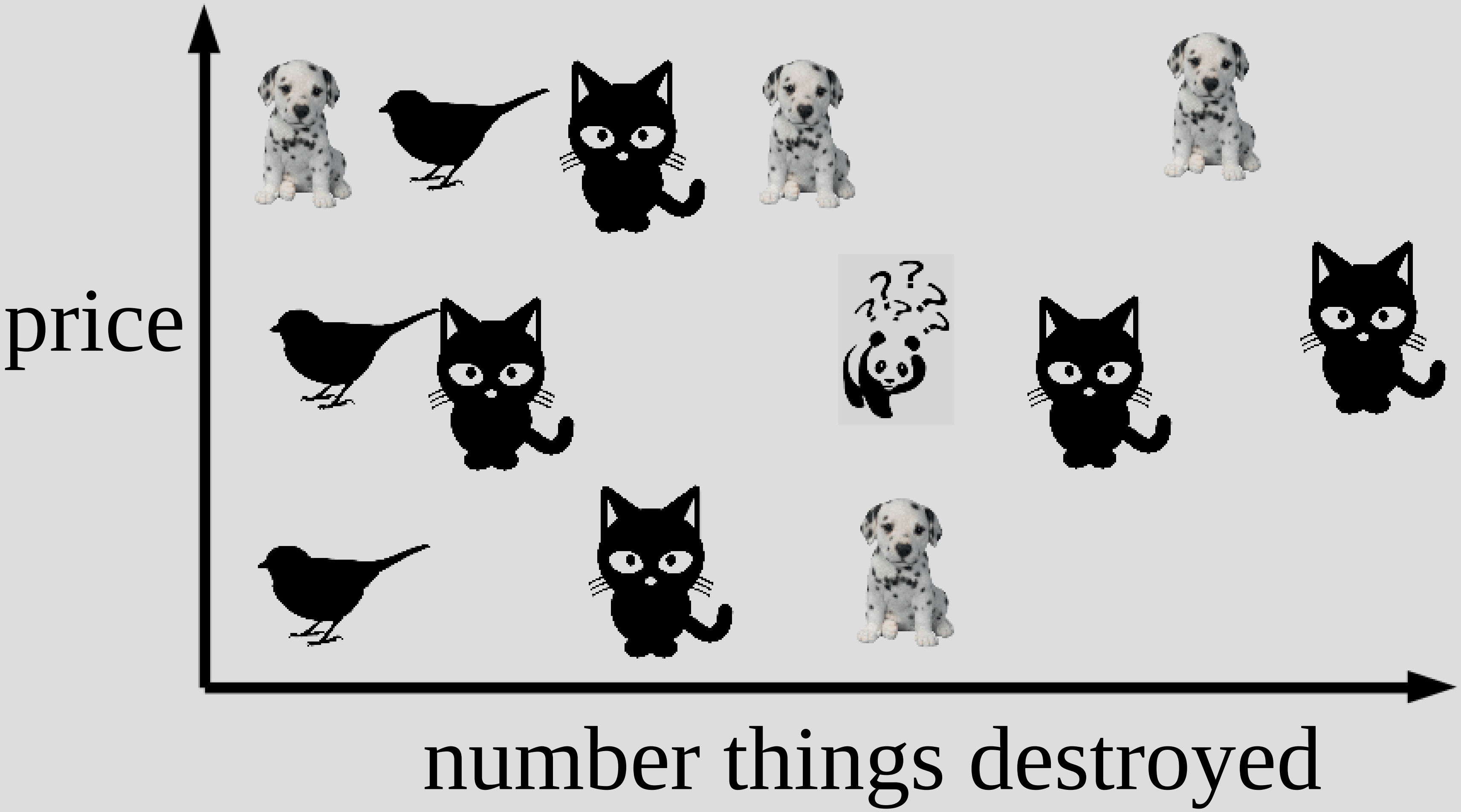
The most simple non-parametric model would be a simple look-up table

1. Store every example seen(and classification)
2. When asked to classify see if example seen before... if not just guess

The accuracy of this model is fairly bad on “new” examples, but it is actually close to being a decent algorithm (despite basic)

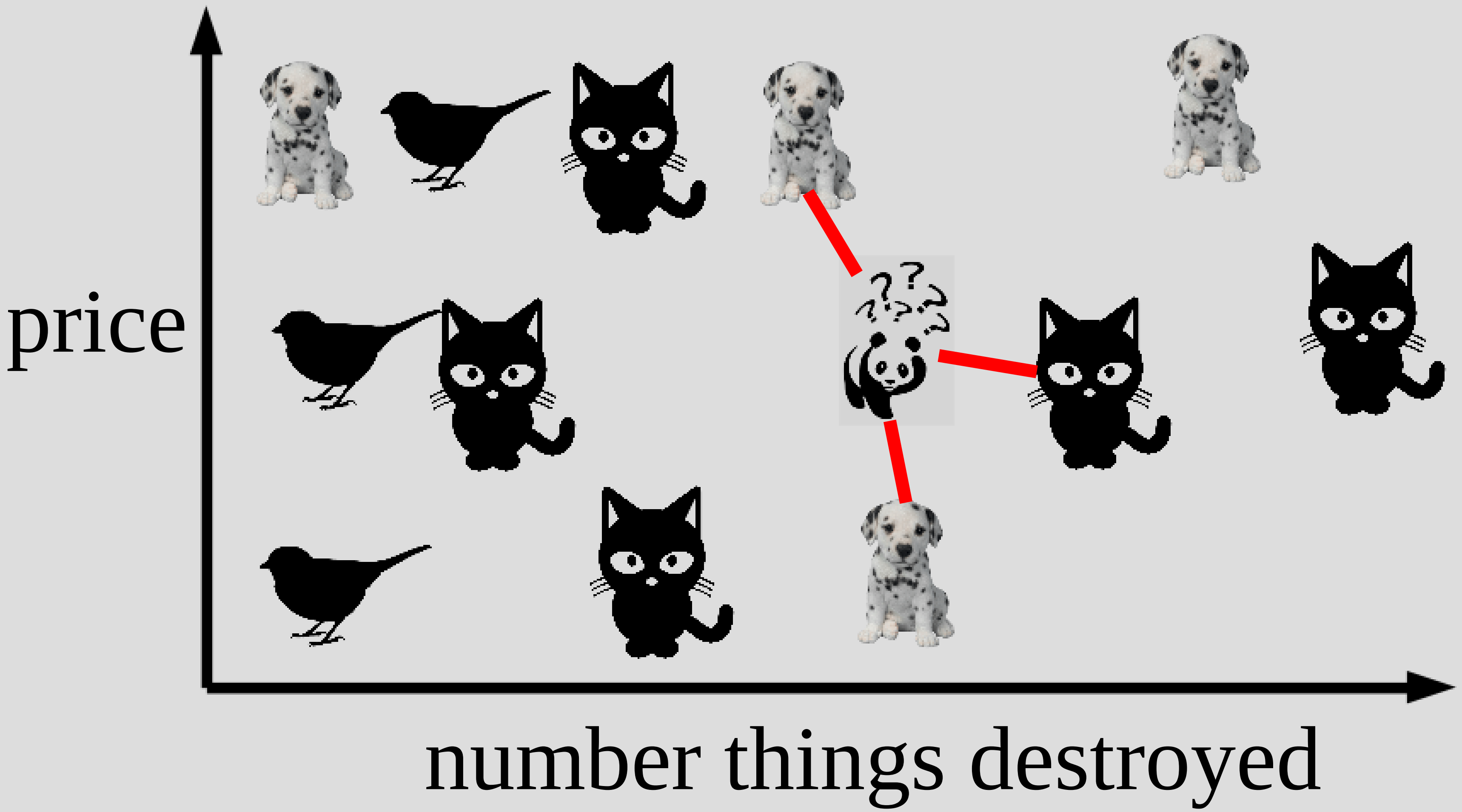
K-nearest neighbor

For example, what is the unknown animal?



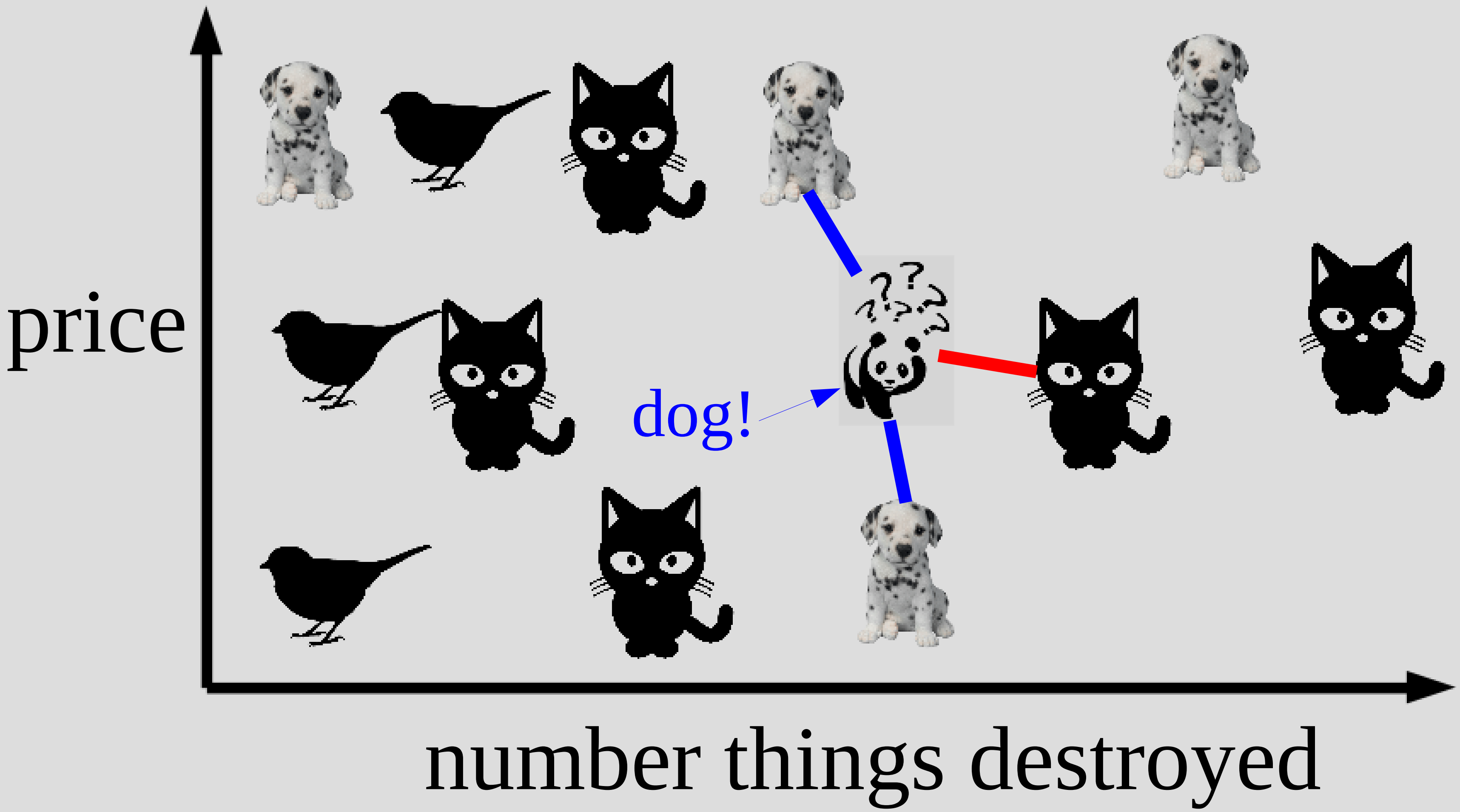
K-nearest neighbor

Assume k=3, so find the 3 nearest neighbors



K-nearest neighbor

Most common neighbor is dog, so our guess



K-nearest neighbor

Rather than just arbitrarily guessing on new examples, we find some “close” examples already seen and use their classification (vote)

To do this we need to define:
1. What is “close”?
2. How many neighbors?

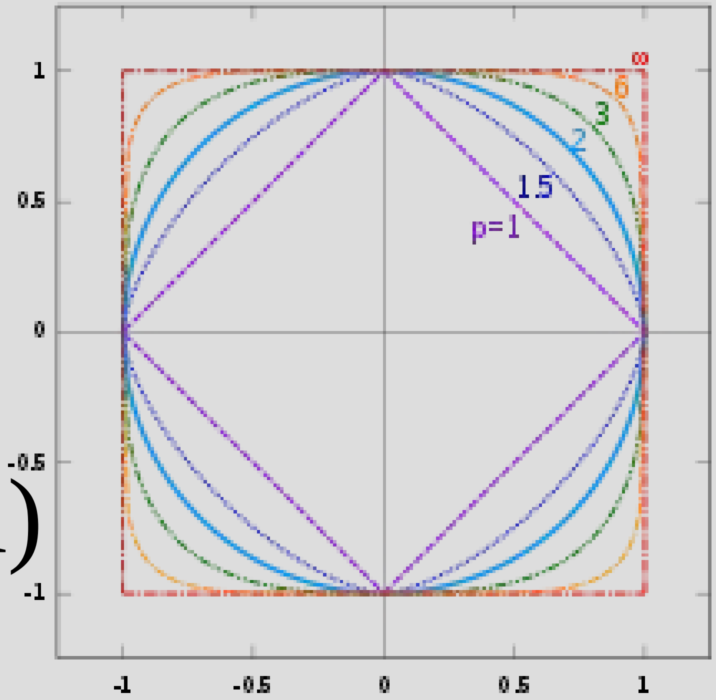
These parameters can be optimized/improved in cross-validation(leave a few out of training)

K-nearest neighbor

1. What is “close”?

Answer: our old friend, the p-norm

$$||x||_p = (|x_1|^p + |x_2|^p + ...)^{1/p}$$



Normally, p=2 (Euclidean distance) is used if similar attributes(e.g. (x,y,z) position)

p=1 (Manhattan) if dissimilar(age, height, etc.)

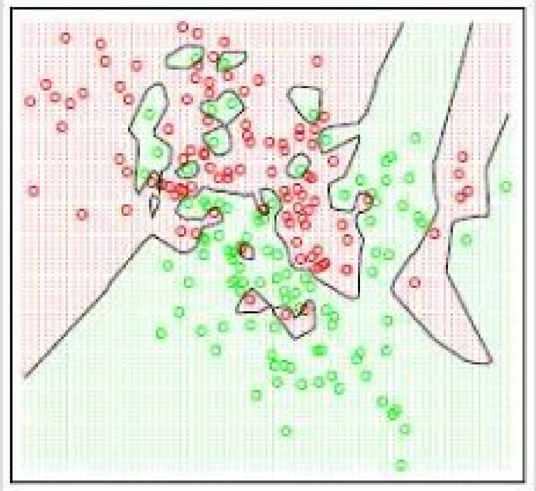
K-nearest neighbor

2. How many neighbors (argument "k")?

Answer: Dunno ¯_(ツ)_/¯

Small k tends to "overfit", large k "underfits"
(What happens when k=all points/examples?)

k=1



k=15

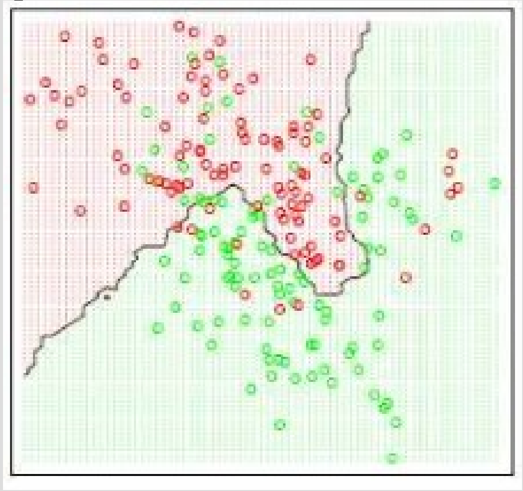


image from:
Hastie, Tibshirani
and Friedman,
Fig 2.2 & 2.3

K-nearest neighbor

Remember this? Let’s use it again

Example	A	B	C	D	E	Ans
1	T	low	big	twit	5	T
2	T	low	small	FB	8	T
3	F	med	small	FB	2	F
4	T	high	big	snap	3	T
5	T	high	small	goog	5	F
6	F	med	big	snap	1	F
7	T	low	big	goog	9	T
8	F	high	big	goog	7	T
9	T	med	small	twit	2	F
10	F	high	small	goog	4	F

replace with numbers

K-nearest neighbor

What problems are there with this?

Example	A	B	C	D	E	Ans
1	1	0	1	0	5	T
2	1	0	0	1	8	T
3	0	1	0	1	2	F
4	1	2	1	2	3	T
5	1	2	0	3	5	F
6	0	1	1	2	1	F
7	1	0	1	3	9	T
8	0	2	1	3	7	T
9	1	1	0	1	2	F
10	0	2	0	3	4	F

K-nearest neighbor

Scale is important, as we are going to find distance

Example	A	B	C	D	E	Ans
1	1	0	1	0	5	T
2	1	0	0	1	8	T
3	0	1	0	1	2	F
4	1	2	1	2	3	T
5	1	2	0	3	5	F
6	0	1	1	2	1	F
7	1	0	1	3	9	T
8	0	2	1	3	7	T
9	1	1	0	1	2	F
10	0	2	0	3	4	F

closer together

more spread out

K-nearest neighbor

Let’s use this normalized-ish version

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

K-nearest neighbor

Suppose we have k=3 and using Manhattan distance (1-norm)

If we saw new data: [1,0,1,0.33,0.2]

We would find the distance to each of the 10 examples

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

K-nearest neighbor

New data = [1,0,1,0.33,0.2]

Example 1=[1,0,1,0   ,0.5]

Distance(new,E1) = 0+0+0+0.33+0.3 = 0.63

Distance(new,E2)=1.6

Distance(new,E3)=2.5

Distance(new,E4)=1.44

Distance(new,E5)=2.97

Distance(new,E6)=1.93

Distance(new,E7)=1.37

Distance(new,E8)=3.17

Distance(new,E9)=1.5

Distance(new,E10)=3.87

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

K-nearest neighbor

New data = [1,0,1,0.33,0.2]

Example 1=[1,0,1,0   ,0.5]

3 nearest

Distance(new,E1) = 0+0+0+0.33+0.3 = 0.63

Distance(new,E2)=1.6

Distance(new,E3)=2.5

Distance(new,E4)=1.44

Distance(new,E5)=2.97

Distance(new,E6)=1.93

Distance(new,E7)=1.37

Distance(new,E8)=3.17

Distance(new,E9)=1.5

Distance(new,E10)=3.87

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

K-nearest neighbor

Since examples 1, 4 and 7 are the closest we see what output is most common among them...

E1=T

E4=T

E7=T

We would guess our new data is also T
(3 votes for, 0 against)

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

K-nearest neighbor

What are the downsides of this approach (assuming you could pick a good “k” and distance measurement/metric)?

1. Scaling issues

What are the downsides of this approach (assuming you could pick a good “k” and distance measurement/metric)?

1. Some issues with scaling (high dimensional input space... i.e. lots of attributes)

2. Computational efficiency... going through all examples for one classification does not scale well...

1. Scaling issues

If you have a large number of inputs/attributes (we had 5 in the previous example)...

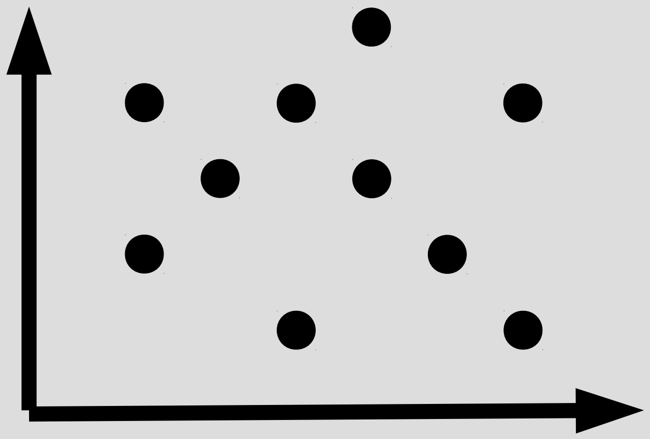
The data tends to be more “spread out” as you simply add more things, thus larger distances

Normalizing does not fix this and it is often hard to do or shouldn’t be done (normalizing makes the assumption all inputs have equal effect on output)

1. Scaling issues

Often called the “curse of dimensionality”

Take a simple case... let N=10 (examples) uniformly distributed in [0,1] and k=1... then what is average distance for 2D space? (Assume distance measure is max difference)



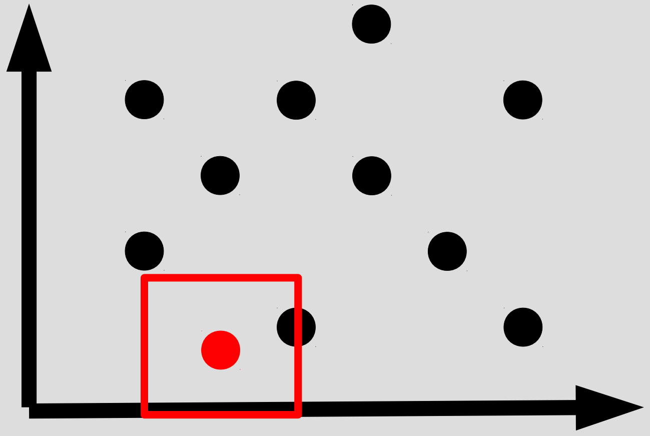
1. Scaling issues

k=1 means we need to find only one point, so on average we'd need the area of the box to be 1/10 the space, as there are 10 examples

Let "L" be the side length of this box, then:

$L^2 = 1/10$ ... or in general (d-dimensions):

$$L^d = k/N$$

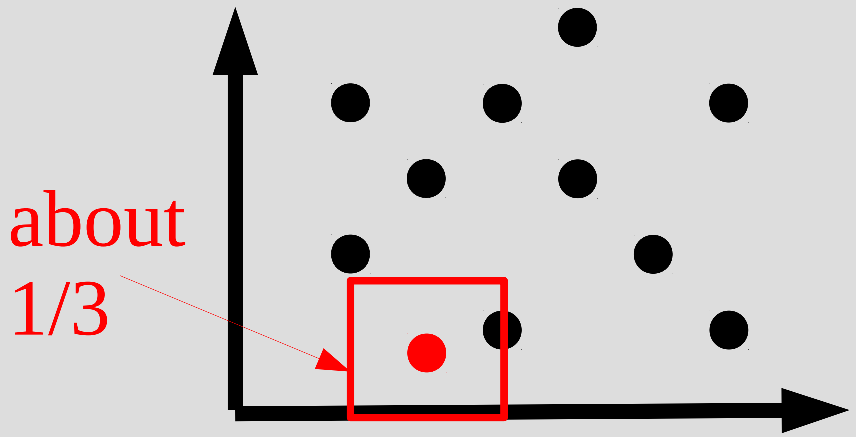


1. Scaling issues

$L^d = k/N$ ...or... $L = (k/N)^{1/d}$

... so in 2-D, the distance (length of box) is $\sqrt{1/10} = 0.316$, if examples in unit square ([0,1])

If we had 5 inputs (like our table), average distance would be: $(1/10)^{1/5} = 0.631$, so the higher the dimension, the more "distant" everything seems (i.e. neighbors not really "near")



2. Complexity issues

The second downside to this approach was that the naive way would be to go through all examples to find nearest

There are two ways we can speed this up:

1. Binary trees (take O(log N)), called <u>k-d trees</u> ("k" is **not** number of neighbors)

2. Locally sensitive hashing (take O(1)-ish) (but this approach is approximate)

2. Complexity issues

With k-d trees, we want to form a full binary tree, so we find a threshold to "split" on

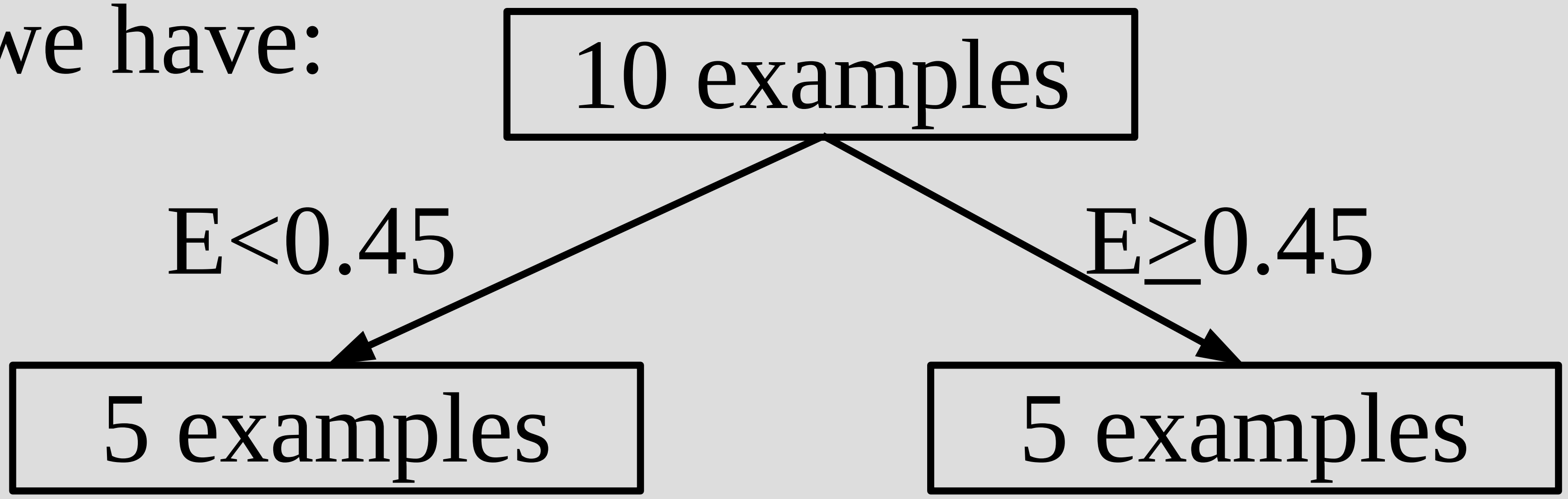
One such is if attribute E < 0.45, we split in half

E < 0.45:
E3, E4, E6, E9, E10
E ≥ 0.45:
E1, E2, E5, E7, E8

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

2. Complexity issues

So we have:



Continue these half splits until only leaves

This lets us make log2(N) decisions before we find a neighbor that is close
... what is the issue with this method?

2. Complexity issues

We might actually need to take both branches

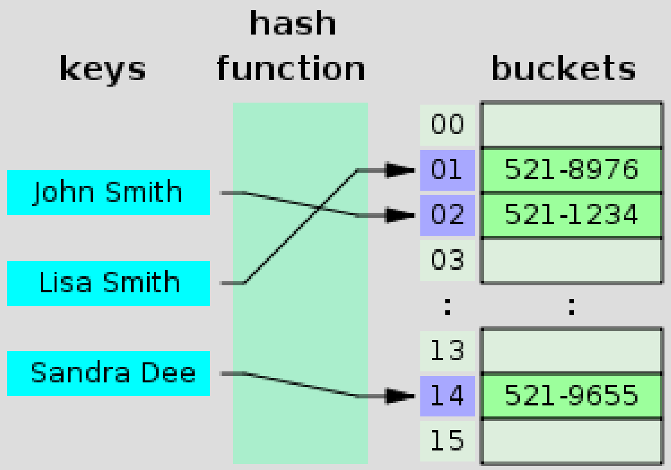
For example, if E=0.45 exactly... it would be unwise to not look at neighbors with E=0.449 (as it might be closest globally)

So you might need to take both branches and evaluate more than just k-neighbors
(some options might not have good half splits as well, especially if discrete data)

2. Complexity issues

The second option is locally sensitive hashing

A hash in general is just a unique-ish identifier (out of this class scope)



We want hashes where the unique-ish ID is “close” if the examples are “near”, these are called <u>locally sensitive hash</u> functions

2. Complexity issues

One pretty simple such hash function is to just pull out a single attribute, so a hash on A:

Key A=1:
E1, E2, E4, E5, E7, E9

Key A=0:
E3, E6, E8, E10

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

2. Complexity issues

Mathematically this is called a projection, as you are reducing/mapping to a smaller set

You could also use hyperplanes/vector to classify either binary or as an integer

Let’s pick the random a random vector:
[-1, -1, 1, 1, 1]

2. Complexity issues

For binary classification, we just look at the sign of the example dot-producted with the vector r=[-1, -1, 1, 1, 1]

$$E1 \cdot r = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0.5 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$
$$= -1 + 0 + 1 + 0 + 0.5$$
$$= 0.5$$

So E1 goes bin positive

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

2. Complexity issues

We end up with the following dot products:

So this hash+vector would put...

Positive bin:
E1, E2, E3,
E6, E7, E8, E10

Negative bin:
E4, E5, E9

E1: 0.5
E2: 0.13
E3: 0.03
E4: -0.03
E5: -0.5
E6: 1.27
E7: 1.9
E8: 1.7
E9: -0.97
E10: 0.4

Example	A	B	C	D	E	Ans
1	1	0	1	0	0.5	T
2	1	0	0	0.33	0.8	T
3	0	0.5	0	0.33	0.2	F
4	1	1	1	0.67	0.3	T
5	1	1	0	1	0.5	F
6	0	0.5	1	0.67	0.1	F
7	1	0	1	1	0.9	T
8	0	1	1	1	0.7	T
9	1	0.5	0	0.33	0.2	F
10	0	1	0	1	0.4	F

2. Complexity issues

If we want more bins, we could pick two random numbers, a and b, where a>b

Then make the bins: $\lfloor \frac{E \cdot r + b}{a} \rfloor$

Pick a=0.5, b=0.2 and we have dot product, so just need include a & b

E1 bin now: $\lfloor \frac{E \cdot r + b}{a} \rfloor = \lfloor \frac{0.5+0.2}{0.5} \rfloor = \lfloor 1.4 \rfloor = 1$

E2 bin now: $\lfloor \frac{E \cdot r + b}{a} \rfloor = \lfloor \frac{0.13+0.2}{0.5} \rfloor = \lfloor 0.66 \rfloor = 0$

... and so on ...

E1: 0.5
E2: 0.13
E3: 0.03
E4: -0.03
E5: -0.5
E6: 1.27
E7: 1.9
E8: 1.7
E9: -0.97
E10: 0.4

2. Complexity issues

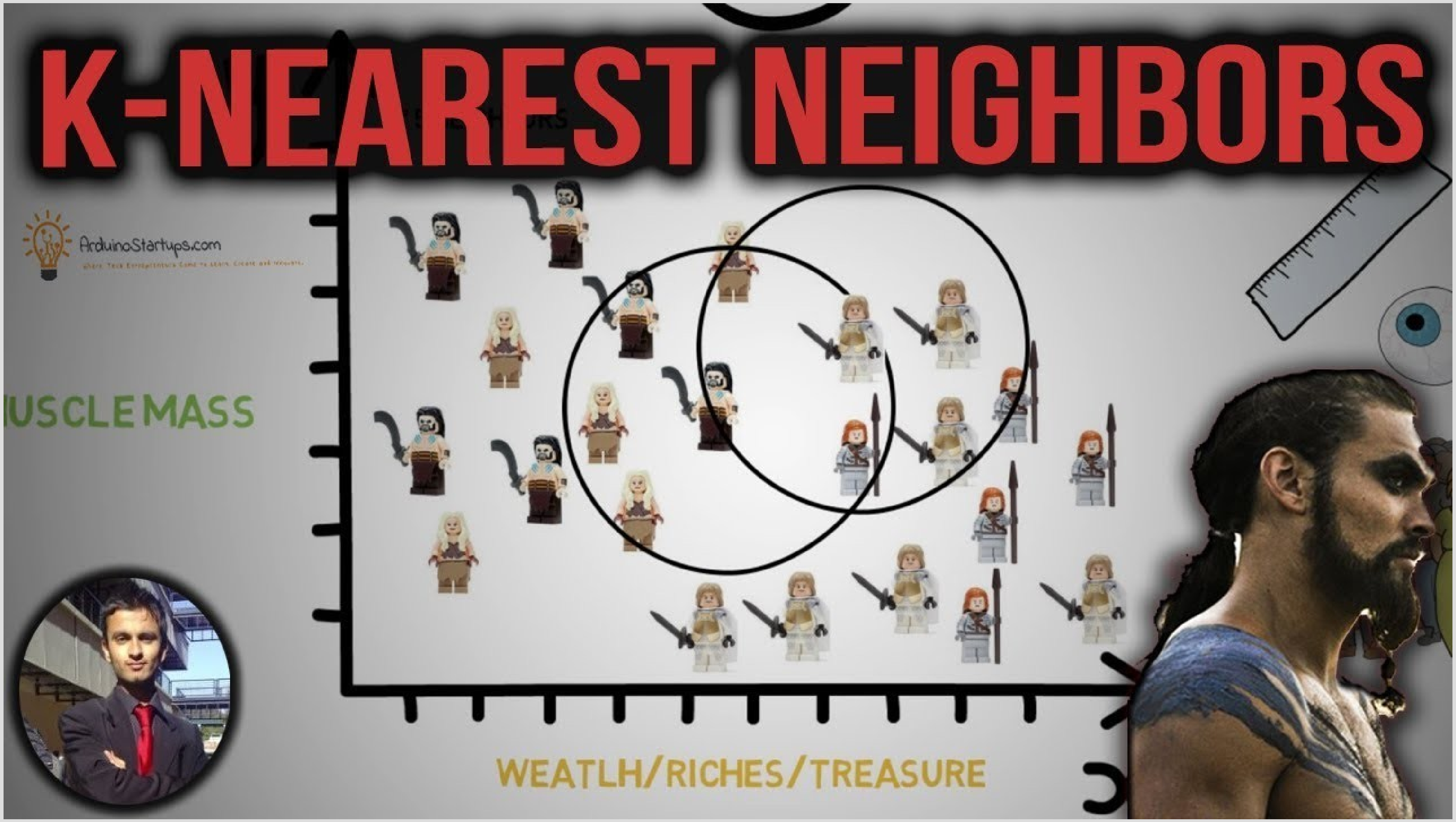
What you would do is then use the same hash function on the new input/query and find the distance only between examples in that bin

This is an approximate approach, and some hashes have theoretical bounds (probability that the nearest will not be in the bin)

You can also have multiple hashes and union the examples across all bins

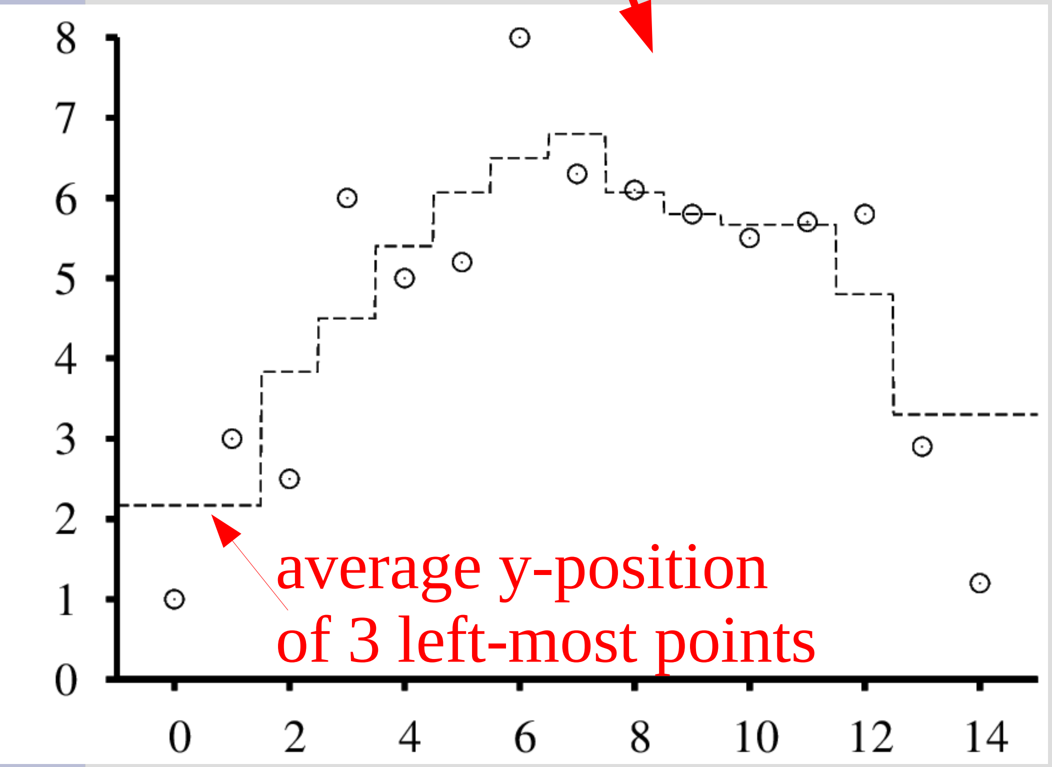
Other resources

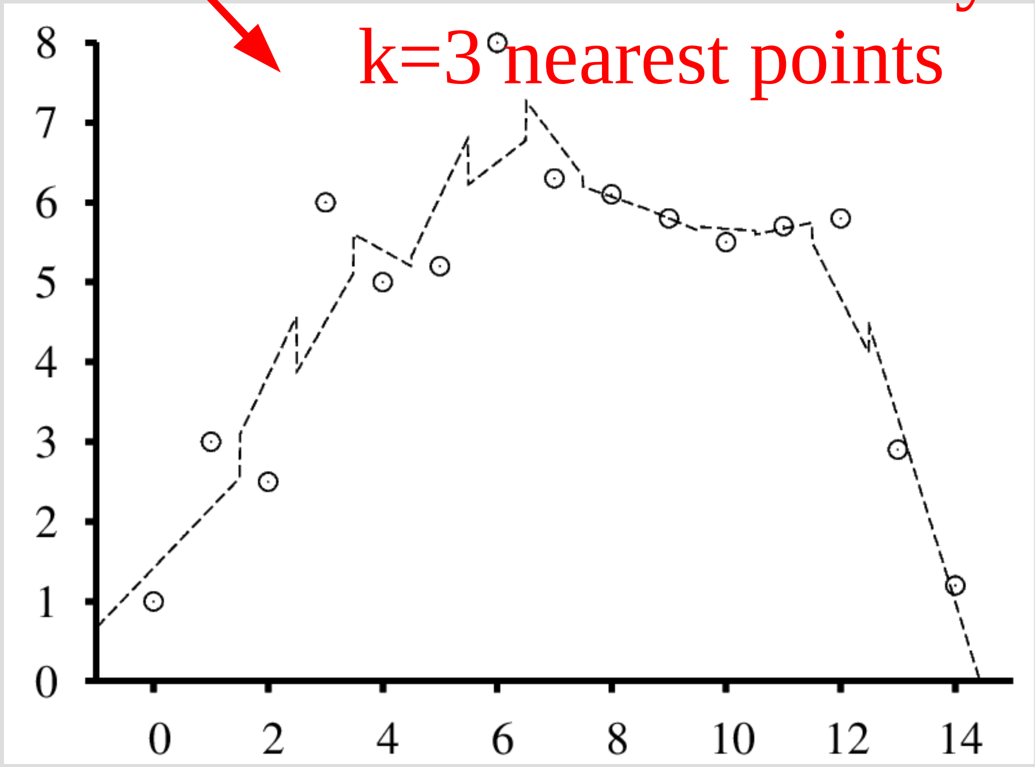
https://www.youtube.com/watch?v=MDniRwXizWo



k-Nearest-Neighbor Regression

You could do regression by taking k-nearest average position or linear regression





k=3 in both figures

k-Nearest-Neighbor Regression

Neither of these are great as not continuous, so let’s investigate a 3rd option

Locally weighted regression uses a weight function to ignore faraway points

(Note: this weight function is called a kernel, which is something completely different than the mathematical definition of a kernel... or even the OS definition in csci... ... .)

k-Nearest-Neighbor Regression

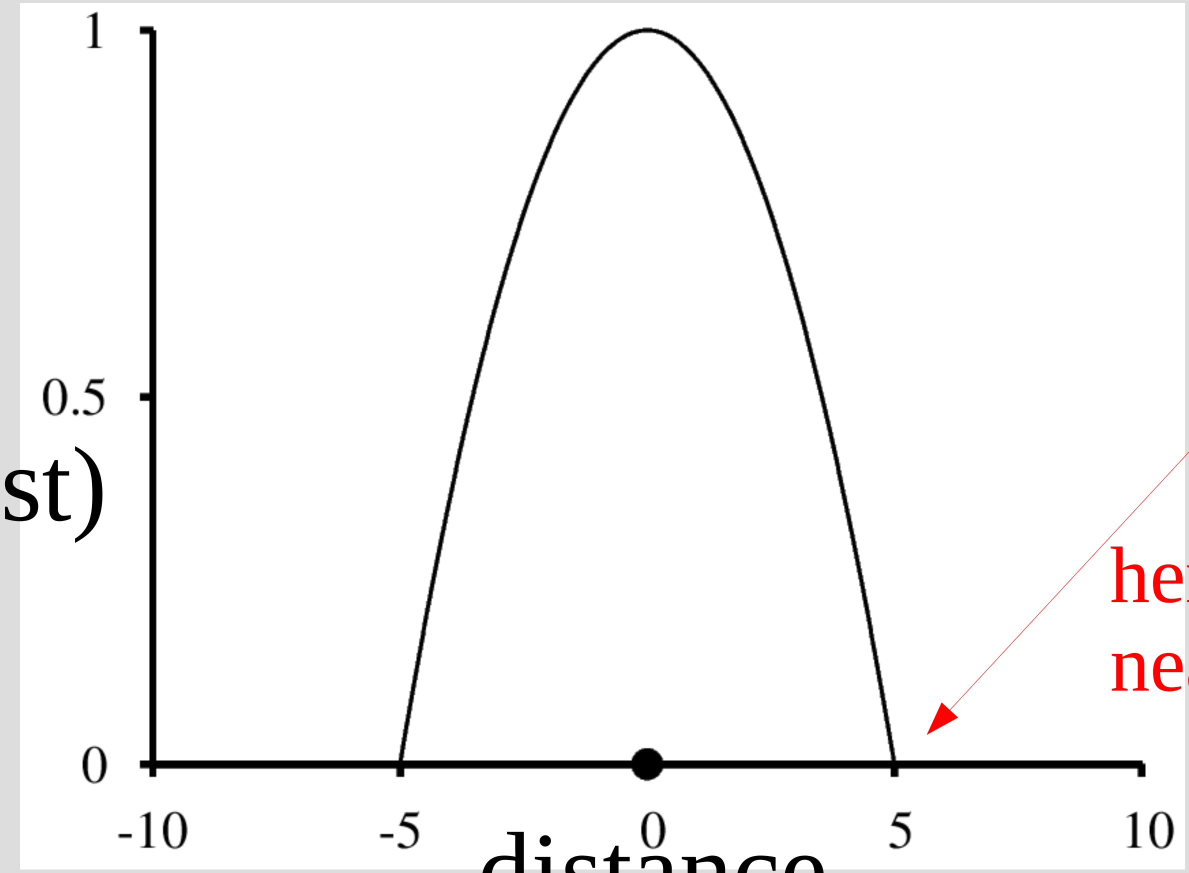
You could use something like a Gaussian as a weighted distance, but that is a bit complex

The weight function/kernel should have:

1. Maximum at distance = 0
2. Symmetric
3. Integral finite/bounded (from $-\infty$ to ∞)

k-Nearest-Neighbor Regression

The book gives this kernel, but the choice is somewhat arbitrary: $K(distance) = \max(0, 1 - (2 \cdot |distance|/k)^2)$



k-Nearest-Neighbor Regression

Once we have the local weight function, we then just solve using gradient descent:

$$w^* = \arg\min_w \sum_j K(Distance(x_j, x_{new}))(y_j - w \cdot x_j)^2$$

sum over all points (many zero)

x_{new} is point you want to find y value for

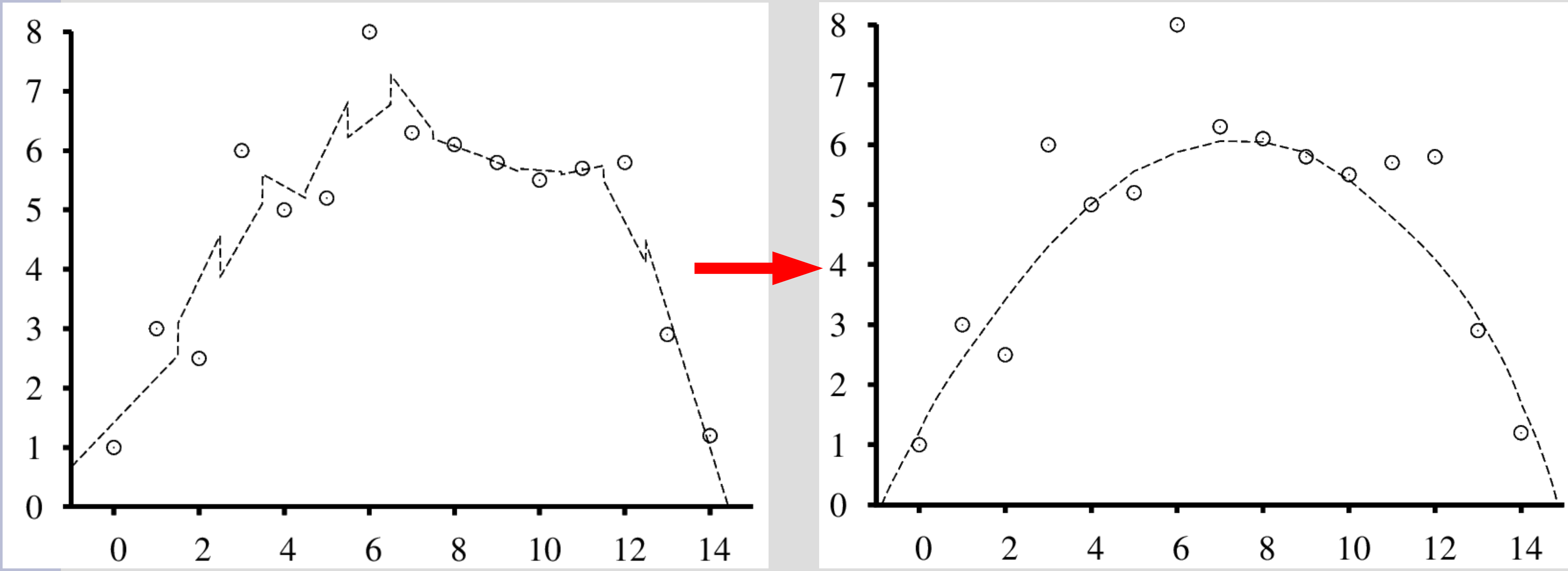
Then your y estimate for x_{new} is:

$$h(x_{new}) = w \cdot x_{new}$$

(most should look familiar from normal linear regression)

k-Nearest-Neighbor Regression

Using this locally weighted regression, we get a much smoother fit (but requires doing gradient descent for each answer)



Parameter Optimization

For all of these there are some parameters that need to be somewhat optimized

What “k” is best?
What distance function to use?
What kernel to use?

All of these can be found by withholding part of training data (cross-validation), and some can be done quite efficiently